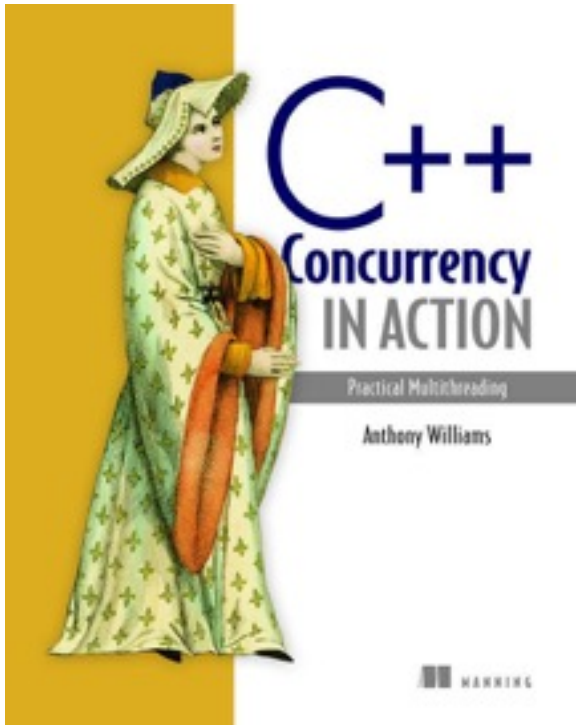
# C++ Korea
## C++ Concurrency in Action Study




### Chapter 06, Designing lock-based concurrent data structures.

C++ Korea 유 주 원 (joowonryoo@gmail.com)

https://github.com/HIPERCUBE/Designing-lock-based-concurrent-data-structure

# This chapter covers

- 동시성을 위한 자료구조 설계는 어떻게 해야하는가?
- 동시성을 위한 자료구조 설계하기 위한 가이드라인
- 동시성 설계가 적용된 자료구조 구현 예시

# This chapter covers

- 동시성을
- 동시성을
- 동시성 실

# What does it mean to design for concurrency?

쓰레드세이프한 자료구조를 설계하는 것

# What does it mean to design for concurrency?

쓰레드세이프한 자료구조를 설계하는 것

직렬화


쓰레드가 데이터에 접근할때 동시에 접근하면 안된다.
=> 순차적으로 접근하게 해야함

# 동시성을 위한 자료구조 설계 가이드라인

고려해야할 사항
- 접근이 안전해야 한다.
- 진정한 동시 접근이 가능하도록 보장한다.

3장에서 공부한 자료구조를 쓰레드-세이프하게 만드는 방법
- 경쟁상태를 피하기위해 주의해야한다.
- invariant가 손상되는것을 막기 위해 예외들이 어떻게 동작하는지 주의를 기울여야한다.
- 교착상태가 일어나지 않도록 해야한다.

만약에 한 쓰레드가 다른 쓰레드에게 불려도 안전한 함수를 통해 자료구조에 접근할때,
과연 이 함수는 다른 쓰레드들로 부터 불려도 안전할까?
(다른 쓰레드에게 불려도 안전하다면서;;)

# 동시성을 위한 자료구조 설계 가이드라인

자료구조 설계자로서 스스로에게 물어봐야할 질문들

- 잠금의 범위를 연산의 일부만 잠그도록 제한할 수 있는가?
- 자료 구조의 각 부분마다 다른 뮤텍스로 보호받을 수 있게 할 수 있는가?
- 모든 연산들이 같은 수준의 보호를 받을 수 있는가?
- 동시성을 위한 기회를 최대화할 수 있는가?

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

3장에서 보았던 쓰레드 세이프 스택의 소스

```
mutable std::mutex m;
```

```
std::lock_guard<std::mutex> lock(m);
```

기본적인 쓰레드 세이프를 제공하는 방법은
멤버함수를 뮤텍스 m으로 잠그는 것이다.

각 멤버함수 첫번째 라인에 있는 std::lock_guard로 뮤텍스 잠금
=> 한번에 오직한 쓰레드가 데이터에 접근하게된다.

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

```
<_Mutex>
void
unique_lock<_Mutex>::lock()
{
    if (__m_ == nullptr)
        __throw_system_error(EPERM, "unique_lock::lock: references null mutex");
    if (__owns_)
        __throw_system_error(EDEADLK, "unique_lock::lock: already locked");
    __m_->lock();
    __owns_ = true;
}
```

극히 드물지만 mutex를 잠글때 예외가 발생할 수 있음
(mutex에 문제가 있거나, 시스템 자원이 부족)

mutex가 nullptr이 될만한 소스가 없다.
mutex unlock이 실패하지 않는다.
std::lock_guard<>가 mutex가 잠긴채로 있지 않도록 보장한다.
=> 안전하다.

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

```
/* 1 */ data = other.data;
```

data가 복사/이동 할때 예외를 던지거나
충분하지 않은 메모리가 할당될 수 있음

std::stack<>

```
_LIBCPP_INLINE_VISIBILITY
stack& operator=(const stack& __q) {c = __q.c; return *this;}
```

```
_LIBCPP_INLINE_VISIBILITY
stack& operator=(stack&& __q)
    _NOEXCEPT_(is_nothrow_move_assignable<container_type>::value)
    {c = _VSTD::move(__q.c); return *this;}
```

std::stack<> 안전하도록 보장함

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }
```

첫번째로 오버로딩한 pop() 함수
스스로 data가 비었는지 확인하고 empty_stack()에러를 던짐
 => 데이터가 비어있을때 문제가 발생하지 않도록함

empty_stack : std::exception

```
struct empty_stack : std::exception {
    const char* what() const throw;
};
```

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }
```

몇가지 이유로 예외가 발생할 수 있음

- std::make_shared 호출이 새 객체를 메모리에 할당하지 못하고 내부 데이터가 레퍼런스 카운팅을 필요로한 경우
- 반환될 아이템의 복사 생성자 혹은 이동 생성자가 새로 할당된 메모리로 복사/이동 하는 경우

C++ 런타임과 표준 라이브러리에서 메모리 누수를 막고, 제대로 새 객체가 소멸되도록 보장한다

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() {}
    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
        data = other.data;
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }
    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }
    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }
    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

std::make_shared

Defined in header <memory>

```
template< class T, class... Args >
shared_ptr<T> make_shared( Args&&... args );
```

Constructs an object of type T and wraps it in a std::shared_ptr using args as the parameter list for the constructor of T.

**Parameters**

args - list of arguments with which an instance of T will be constructed.

**Return value**

std::shared_ptr of an instance of type T.

**Exceptions**

May throw std::bad_alloc or any exception thrown by the constructor of T. If an exception is thrown, this function has no effect.

```
std::shared_ptr<T> pop() {
    std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
    return res;
}
```

몇가지 이유로 예외가 발생할 수 있음

- std::make_shared 호출이 새 객체를 메모리에 할당하지 못하고 내부 데이터가 레퍼런스 카운팅을 필요로한 경우
- 반환될 아이템의 복사 생성자 혹은 이동 생성자가 새로 할당된 메모리로 복사/이동 하는 경우

C++ 런타임과 표준 라이브러리에서 메모리 누수를 막고, 제대로 새 객체가 소멸되도록 보장한다

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }
```

결과 반환으로 예외를 던지지 못하게 보장함.
exception-safe 하다.

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }
```

두번째로 오버로딩한 pop() 함수
이것도 첫번째 pop() 함수와 비슷하다.
복사 할당 혹은 이동 할당 연산자가 예외를 던질수있다.

data.pop()이 호출되기 전까지 자료 구조를 수정하지 않는것은
예외 발생을 막는것이다.

=> 이 pop() 함수도 exception-safe 하다.

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
```

empty() 함수는 데이터를 수정하지 않는다.

=> 이 empty() 함수도 exception-safe 하다.

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

모든 멤버 함수가 데이터를 보호하기 위해 std::lock_guard<> 사용

=> 여러 쓰레드가 스택의 멤버함수를 호출해도 안전

생성자와 소멸자는 예외

객체는 오직 한번 생성되고 한번 소멸된다.

완전히 생성되지 않거나 파괴된 객체의 멤버함수를 호출하지 않도록 해야한다.

=> 객체가 완벽하기 생성되기 전에는 다른 쓰레드가 접근하지 못하도록 해야함

=> 스택이 소멸되기 전에 접근을 중단시켜야한다.

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

직렬화로 인해 한번에 오직 한 쓰레드만 스택에서 일을 할 수 있다.

직렬화는 어플리케이션의 성능을 제한시킴
한 쓰레드가 잠금을 기다리는 동안에는 아무것도 할 수 없다.
스택은 추가될 아이템을 기다리는 방법을 제공하지 않음.
쓰레드가 기다려야한다면, 주기적으로 empty()을 호출하거나,
그냥 pop()을 호출하고, empty_stack 예외를 캐치해야함.

대기하는 쓰레드는 데이터를 확인하면서 자원을 소비하거나,
사용자가 외부 대기 및 알림코드(ex 상태변수)를 작성해야함

4장의 큐는 자료구조 안에서 상태변수를 이용해서 스스로 대기함

https://github.com/CppKorea/CppConcurrencyInAction/wiki/Chapter-04,-Synchronizing-concurrent-operations
조건변수를 사용해 특정 조건 기다리기

# Listing6.1 잠금을 사용한 쓰레드세이프 스택

```
struct empty_stack : std::exception {
    const char* what() const throw;
};

template<typename T>
class threadsafe_stack {
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack() { }

    threadsafe_stack(const threadsafe_stack& other) {
        std::lock_guard<std::mutex> lock(other.m);
/* 1 */ data = other.data;
    }

    threadsafe_stack& operator=(const threadsafe_stack) = delete;

    void push(T new_value) {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }

    std::shared_ptr<T> pop() {
        std::lock_guard<std::mutex> lock(m);
/* 2 */ if (data.empty()) throw empty_stack();
/* 3 */ std::shared_ptr<T> const res(std::make_shared(std::move(data.top())));
/* 4 */ data.pop();
        return res;
    }

    void pop(T& value) {
        std::lock_guard<std::mutex> lock(m);
        if (data.empty()) throw empty_stack();
/* 5 */ value = std::move(data.top());
/* 6 */ data.pop();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

직렬화로

직렬화는
한 쓰레드
스택은 추
쓰레드가
그냥 pop

대기하는
사용자가

4장의 큐

https://github
조건변수를 사용하

t-operations

## 조건변수를 사용해 특정 조건 기다리기

> Waiting for a condition with condition variables

C++ 표준 라이브러리는 2개의 조건 변수 `std::condition_variable` , `std::condition_variable_any` 를 제공한다. 모두 `<condition_variable>` 라이브러리 헤더에 선언되었다. 모두 적절한 동기화를 제공하기 위해 뮤텍스와 함께 작동할 필요가 있다. 전자는 `std::mutex` 와 작업하는 것으로 제한되는 것에 반하여 후자는 뮤텍스와 같게되는 최소 기준과 작업할 수 있다. 그래서 `_any` 접미사가 붙는다. 따라서 `std::condition_variable_any` 는 좀더 일반적이고, 규모, 성능, 시스템 리소스에 대해서 추가 비용의 가능성이 있다. 그래서 `std::condition_variable` 는 유연성이 필요하지 않으면 선호된다.

그래서 어떻게 데이터가 처리될때가지 잠긴 상태에서 대기중인 쓰레드 예제를 처리하기 위해 `std::condition_variable` 을 사용하는가? 첫째로, 당신은 두 개의 스레드 사이의 데이터를 전달하는 데 사용될 큐를 가지고있다.데이터가 준비되면 쓰레드는 `std::lock_guard` 을 사용하여 큐를 보호하고 데이를 큐에 삽입한다. 그리고 대기하고 있는 쓰레드에게 알리기 위해 `std::condition_variable` 에 있는 `notify_one()` 을 호출한다.

멘스의 다른쪽에 처리중인 쓰레드가 있습니다. 쓰레드는 첫번째로 뮤텍스를 잠그나, 이번에는 `std::lock_guard` 보다 `std::unique_lock` 이 좋은데 잠시 후 이유를 알 수 있다. 쓰레드는 `std::condition_variable` 에 있는 `wait()` 를 호출하고 나서, 잠금 객체와 대기를 위한 람다 표현 조건 통과한다. 람다 함수는 다른 표현의 일부로 익명 함수를 작성할 수 있도록 C++11에서 추가되었고 `wait()` 와 같은 표준 라이브러리 함수에 조건을 지정하는데 적합하다. 이경우 단순한 람다함수는 data_queue가 비었는지 체크한다—어떤 데이터가 큐안에서 처리될 준비가 되었다면

람다 함수는 상세한 설명을 appendix A를 참조

`wait()` 의 구현은 조건을 검사하고 (제공된 람다함수를 호출하여) 조건을 만족하면 true를 만족하지 않으면 false를 반환한다. `wait()` 는 뮤텍스를 해제하고 대기상태또는 잠겨있는 쓰레드에 넣는다. 조건변수는 데이터가 준비된 쓰레드에서 `notify_one()` 을 호출하여 통보하면, 잠든 상태(잠기지 않은)에서 쓰레드는 깨어나 뮤텍스 잠금을 다시 획득하고, 상태를 다시 체크하고 조건이 만족한다면 잠김 뮤텍스와 wait()에서 반환된다. 조건을 충족하지 않으면 쓰레드는 뮤텍스 잠금을 해제하고, 대기를 다시 시작한다. 이것이 `std::lock_guard` 대신에 `std::unique_lock` 이 필요한지에 대한 이유다. —대기중인 쓰레드는 대기하고 잠기는 동안 다시 뮤텍스 잠금을 해제해야 하고. `std::lock_guard` 은 유연함을 제공하지 않는다. 뮤텍스는 쓰레드가 잠든동안 잠근 상태를 유지한다면 , 데이터 준비 쓰레드는 큐에 아이템을 추가할때 뮤텍스 잠금을 할 수 없을 것이고, 대기중인 쓰레드는 조건을 충족하는 것을 볼 수 없을 것이다.

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

```
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

data_cond.notify_one(), wait_and_pop()을 제외하면 Listing6_1과 비슷

try_pop() 메소드들은 큐가 비어있어도 예외를 던지지 않는것을 제외하면 비슷
- 예외를 던지지 않는 대신에 데이터가 있는지 알려주는 bool 값을 반환
- 데이터가 없는 경우에는 NULL 포인터를 반환

```
    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }
```

```
    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
```

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

```
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

wait_and_pop() 함수들은 대기문제의 해결책

empty()를 계속 호출하는것보다 대기하는 쓰레드가 wait_and_pop()을 한번 호출하고 자료구조가 상태변수로 기다려 주는것이 낫다.

data_cond.wait()는 큐가 적어도 하나의 원소를 가질때까지 반환하지 않는다.
=> 비어있는 큐에서 가능한지, 그리고 뮤텍스의 잠금으로 계속 보호 받는지 걱정할 필요 X

```
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }
```

```
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
```

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

여러 쓰레드가 아이템이 푸시되고 있는 큐를 기다릴때,
오직 한 쓰레드가 data_cond.notify_one()에 의해 깨어난다.
하지만 쓰레드가 wait_and_pop()에서 새 std::shared_ptr<> 생성에서 예외를 던지면
다른 쓰레드들은 깨어나지 않는다.

```cpp
    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }
```

```cpp
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
```

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

여러 쓰레드가 아이템이 푸시되고 있는 큐를 기다릴때,
오직 한 쓰레드가 data_cond.notify_one()에 의해 깨어난다.
하지만 쓰레드가 wait_and_pop()에서 새 std::shared_ptr<> 생성에서 예외를 던지면
다른 쓰레드들은 깨어나지 않는다.

condition_variable::wait()

```cpp
#ifndef _LIBCPP_HAS_NO_THREADS
<_Predicate>
void
condition_variable::wait(unique_lock<mutex>& __lk, _Predicate __pred)
{
    while (!__pred())
        wait(__lk);
}
```

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

## Solution 1.

data_cond.notify_all()
모든 쓰레드를 깨우지만, 대부분 큐가 비어있는것을 확인하고 다시 자러감

### std::condition_variable::notify_all

`void notify_all();` (since C++11)

Unblocks all threads currently waiting for `*this`.

**Parameters**

(none)

**Return value**

(none)

**Exceptions**

noexcept specification: `noexcept`

**Notes**

The effects of notify_one()/notify_all() and wait()/wait_for()/wait_until() take place in a single total order, so it's impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock.

출처 : cppreference.com

[MSDN 참고]
L condition_variable::notify_all : condition_variable 개체를 기다리는 모든 스레드를 차단해제합니다.
L condition_variable::notify_one : condition_varibale 개체를 기다리는 스레드 중 하나를 차단해제 합니다.

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

## Solution 2.

data_cond.notify_one()
wait_and_pop()이 예외를 던질때 notify_one()을 호출하도록한다.
notify_one()을 호출함으로써 다른 쓰레드는 저장된 값을 탐색할 수 있다.

### std::condition_variable::notify_one

`void notify_one();` (since C++11)

If any threads are waiting on `*this`, calling notify_one unblocks one of the waiting threads.

**Parameters**

(none)

**Return value**

(none)

**Exceptions**

noexcept specification: `noexcept`

**Notes**

The effects of notify_one()/notify_all() and wait()/wait_for()/wait_until() take place in a single total order, so it's impossible for notify_one() to, for example, be delayed and unblock a thread that started waiting just after the call to notify_one() was made.

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock. However, some implementations (in particular many implementations of pthreads) recognize this situation and avoid this "hurry up and wait" scenario by transferring the waiting thread from the condition variable's queue directly to the queue of the mutex within the notify call, without waking it up.

출처 : cppreference.com

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

# Listing6.2 lock과 상태변수를 사용한 쓰레드세이프 큐

**Solution 3.**

std::shared_ptr<> 생성자를 push() 호출할때로 옮기고
std::shared_ptr<> 인스턴스 대신 데이터값을 직접 저장

std::shared_ptr<>을 내부 std::queue<> 밖에서 복사를 하게 되면
예외를 던지지않아, wait_and_pop()은 안전하다.

# Listing6.3 std::shared_ptr<> 인스턴스 보유한 쓰레드세이프 큐

```
#include <__mutex_base>
#include <queue>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T>> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 1 */ value = std::move(*data_queue.front());
        data_queue.pop();
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return false;
/* 2 */ value = std::move(*data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 3 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return std::shared_ptr<T>();
/* 4 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    void push(T new_value) {
/* 5 */ std::shared_ptr<T> data(std::make_shared<T>(std::move(new_value)));
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

새 데이터를 받기 위해 래퍼런스 변수를 받아오는 pop() 함수들은 저장된 포인터를 역참조
std::shared_ptr<> 인스턴스를 반환하는 pop() 함수들은 호출자에게 반환하기전에 큐에서 찾음

```
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 1 */ value = std::move(*data_queue.front());
        data_queue.pop();
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return false;
/* 2 */ value = std::move(*data_queue.front());
        data_queue.pop();
        return true;
    }
```

```
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 3 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return std::shared_ptr<T>();
/* 4 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }
```

# Listing6.3 std::shared_ptr<> 인스턴스 보유한 쓰레드세이프 큐

```
#include <__mutex_base>
#include <queue>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T>> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 1 */ value = std::move(*data_queue.front());
        data_queue.pop();
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return false;
/* 2 */ value = std::move(*data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 3 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return std::shared_ptr<T>();
/* 4 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    void push(T new_value) {
/* 5 */ std::shared_ptr<T> data(std::make_shared<T>(std::move(new_value)));
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

## Advantage

새 인스턴스 할당이 push()의 잠금 밖에서 가능
Listing6.2에서는 pop()이 잠겨있는 동안 했음
메모리 할당이 꽤 비싼 연산이기 때문에, 큐의 성능면에서 이득이다.
뮤텍스가 잡고있는 시간을 줄여주고, 다른 쓰레드가 그동안 다른 작업들을 수행할 수 있도록 해준다.

## Listing6.3

```
    void push(T new_value) {
/* 5 */ std::shared_ptr<T> data(std::make_shared<T>(std::move(new_value)));
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
```

## Listing6.2

```
    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }
```

# Listing6.3 std::shared_ptr<> 인스턴스 보유한 쓰레드세이프 큐

```
#include <__mutex_base>
#include <queue>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T>> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 1 */ value = std::move(*data_queue.front());
        data_queue.pop();
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return false;
/* 2 */ value = std::move(*data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
/* 3 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
            return std::shared_ptr<T>();
/* 4 */ std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    void push(T new_value) {
/* 5 */ std::shared_ptr<T> data(std::make_shared<T>(std::move(new_value)));
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

## Limitation

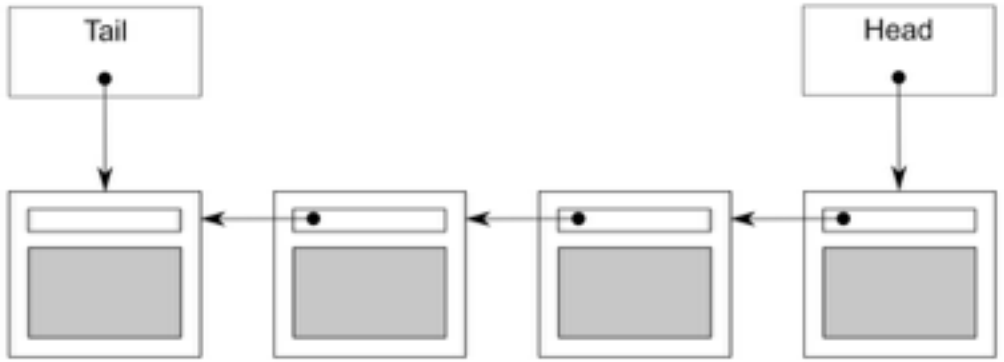
스택예제처럼 한 뮤텍스를 전체 자료구조를 보호하기 위해 사용하는 것은 큐가 지원하는 동시성을 제한
여러 쓰레드들이 큐의 멤버 함수들에서 막히면, 한번에 한 쓰레드만 일할 수 있다.
std::queue<> 사용때문에 동시성을 최대화 시키지 못함

# Listing6.4 쓰레드세이프 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

**싱글 쓰레드 큐 구현**

- head 포인터는 리스트의 첫번째 아이템을 참조한다.
- 각 아이템은 다음 아이템을 가리킴
- 데이터를 꺼낼때는 헤드를 이동하고 꺼낸다.
- 아이템들은 tail쪽에 추가된다.
- tail 포인터는 마지막 아이템을 참조한다.
- 새 아이템을 추가될때
  1. 마지막 아이템의 next 포인터를 새 아이템으로 업데이트
  2. tail 포인터가 새 아이템을 참조
- 싱글 쓰레드만 지원하기 때문에, try_pop() 함수만 있고, wait_and_pop() 함수는 없다.



# Listing6.4 싱글쓰레드 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

std::unique_ptr<node> 노드들을 관리하기 위해 사용
이렇게 사용하면 delete를 작성하지 않아도 자동으로 메모리 회수됨

```
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };
```

head와 tail로 소유권 체인 관리

```
/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;
```

# Listing6.4 싱글쓰레드 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

**이 큐를 멀티쓰레드에서 사용한다고 했을때 문제점은?**

# Listing6.4 싱글쓰레드 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

push()가 head와 tail을 수정한다.

```
    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
```

head와 tail에 각각 뮤텍스를 따로 가지고 있어야한다.
하나는 head 보호, 다른 하나는 tail 보호

```
/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;
```

# Listing6.4 싱글쓰레드 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

push()와 pop() 둘다 노드의 next 포인터에 접근

```
    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }
```

```
    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
```

# Listing6.4 싱글쓰레드 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

만약에 큐에 한 아이템만 있다면 어떻게 될까?

# Listing6.4 싱글쓰레드 큐 구현

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };

/* 1 */
    std::unique_ptr<node> head;
/* 2 */
    node* tail;

public:
    queue() { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }

    void push(T new_value) {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_tail = p.get();
        if (tail) {
/* 4 */     tail->next = std::move(p);
        } else {
/* 5 */     head = std::move(p);
        }
/* 6 */ tail = new_tail;
    }
};
```

**만약에 큐에 한 아이템만 있다면 어떻게 될까?**

head == tail
head->next == tail->next

push()와 try_pop() 모두 같은 뮤텍스로 잠궈야 한다.

=> 결국 세밀한 잠금을 하지 못함

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

**더미노드**를 미리 할당함으로써 해결 가능

더미노드를 할당하기 때문에 큐에는 적어도 1개 이상의 노드(head 이면서 tail)가 있다.

이제 빈 큐의 head와 tail은 NULL이 아니라 더미 노드를 가리킨다.

큐가 비어있을때 try_pop()이 head->next에 접근하지 않기 때문에 안전

큐에 노드를 추가한다면, head와 tail은 각가 다른 노드 가리킴

=> head->next와 tail->next사이에 경쟁이 없게된다.

단점 : 더미노드를 위해 포인터로 데이터를 저장하는 추가의 인다이렉션 발생

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

```cpp
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

## Listing6.5

```cpp
    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }
```

더미노드가 있어서 head가 절대 NULL 아님

## Listing6.4

```cpp
    std::shared_ptr<T> try_pop() {
        if (!head)
            return std::shared_ptr<T>();
        std::shared_ptr<T> const res(std::make_shared<T>(std::move(head->data)));
        std::unique_ptr<node> const old_head = std::move(head);
/* 3 */ head = std::move(old_head->next);
        return res;
    }
```

```cpp
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

**Listing6.5**

```cpp
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
```

**Listing6.4**

```cpp
    struct node {
        T data;
        std::unique_ptr<node> next;

        node(T data_) : data(std::move(data_)) { }
    };
```

node가 포인터로 데이터를 저장하기 때문에,
새 T 인스턴스를 만드는것보다 낫다.

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

```
    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
```

힙에 T 인스턴스를 먼저 만들고 std::shared_ptr<>로 가져온다.
(두번째 메모리 할당의 오버헤드를 피하기 위해 std::make_shared 사용)

새로 만드는 노드는 더미노드라서 생성자에게 new_value를 넘겨줄 필요 없다.

대신에 전 노드에 new_value를 복사해서 넣어준다.

```
    queue() : head(new node), tail(head.get()) { }
```

더미 노드는 생성자에서 만들어 준다.

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

```
    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
```

```
    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }
```

push()는 head가 아니라 tail에 접근한다.
try_pop()은 head와 tail에 모두 접근하지만, tail은 초기비교에만 필요하고, 잠금은 일시적
더미노드때문에 try_pop()과 push()가 같은 노드에서 연산하지 않아서 뮤텍스가 필요X
 => head용 뮤텍스 하나랑 tail용 하나만 있으면 된다.

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

```
    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
```

뮤텍스는 tail에 대한 모든 접근에서 잠겨야한다.
(새 노드가 할당된후와 현재 tail 노드에 데이터를 할당하기전에 뮤텍스를 잠궈야한다.)
잠긴 다음에 함수가 끝날때까지 유지시켜야한다.

# Listing6.5 싱글쓰레드 큐 구현 with 더미노드

```
#include <memory>

template<typename T>
class queue {
private:
    struct node {
/* 1 */ std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::unique_ptr<node> head;
    node* tail;

public:
    /* 2 */
    queue() : head(new node), tail(head.get()) { }

    queue(const queue& other) = delete;

    queue& operator=(const queue& other) = delete;

    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }

    void push(T new_value) {
/* 7 */ std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
/* 8 */ std::unique_ptr<node> p(new node);
/* 9 */ tail->data = new_data;
        node* const new_tail = p.get();
        tail->next = std::move(p);
        tail = new_tail;
    }
};
```

```
    std::shared_ptr<T> try_pop() {
/* 3 */ if (head.get() == tail) {
            return std::shared_ptr<T>();
        }
/* 4 */ std::shared_ptr<T> const res(head->data);
        std::unique_ptr<node> old_head = std::move(head);
/* 5 */ head = std::move(old_head->next);
/* 6 */ return res;
    }
```

먼저 head 뮤텍스를 잠그고, head 사용이 끝날때 까지 기다려야함.
뮤텍스는 어떤 쓰레드가 pop할지 결정한다. 그래서 먼저 잠궈야한다.
head가 변경되면, 뮤텍스의 잠금을 풀어야한다.
결과를 반환할때 잠글 필요는 없다.
한번만 tail에 접근하기 때문에, 읽는 시간 동안 뮤텍스를 얻을 수 있다.
함수를 감싸는것이 가장 좋다.
전부다 head를 사용하는 소스라서, 함수를 감싸는게 명확하다.

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

- tail->next == nullptr
- tail->data == nullptr
- head == tail 은 리스트가 비었음을 의미한다.
- 1개의 원소가 있을때는 head->next == tail 이다.
- 리스트의 각 노드 x에 대하여,

  if (x != tail) x->data는 T 인스턴스를 가리키고, x->next는 리스트의 다음노드르러 가리킴
- head의 next 노드를 따라가다보면 tail이 나온다.

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

```
void push(T new_value) {
    std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    node* const new_tail = p.get();
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    tail->data = new_data;
    tail->next = std::move(p);
    tail = new_value;
}
```

트집잡을만한게 없음
수정은 tail_mutex에 의해 보호받고, invariant 상태를 유지한다.

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
std::shared_ptr<T> try_pop() {
    std::unique_ptr<node> old_head = pop_head();
    return old_head ? old_head : std::shared_ptr<T>();
}
```

```cpp
std::unique_ptr<node> pop_head() {
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if (head.get() == get_tail()) {
        return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
}
```

tail_mutex로 tail 읽기만 보호하면 되는게 아니라
head를 읽을때 경쟁상태가 발생하지 않도록 해야함

mutex를 가지고 있지 않으면,
try_pop()을 호출하는 쓰레드와 push()를 동시에 호출하는 쓰레드는 정의된 작업순서 없음
각 멤버함수를 뮤텍스로 잠근다면, 서로 다른 뮤텍스로 잠궈야 한다.

같은 데이터에 접근한 가능성이 있음
큐의 데이터들은 push()로 들어갔다.
쓰레드들이 작업순서 정의 없이 같은 데이터에 접근하면 경쟁상태와 undefined behavior 발생

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
node* get_tail() {
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    return tail;
}
```

```cpp
void push(T new_value) {
    std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    node* const new_tail = p.get();
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    tail->data = new_data;
    tail->next = std::move(p);
    tail = new_value;
}
```

감사하게도 get_tail()의 tail_mutex 잠금이 모든것을 해결

get_tail()의 호출은 push()의 호출과 같이 같은 뮤텍스를 잠그기 때문에,
두 호출 사이에 순서가 정의되어있다.

tail의 전 값을 볼때는 get_tail()의 호출이 push()전에 일어나고,
tail의 새 값을 보고 전 tail의 값을 새 데이터에 넣을때는, push() 이후에 발생

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

```cpp
std::unique_ptr<node> pop_head() {
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if (head.get() == get_tail()) {
        return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
}
```

get_tail()의 호출이 head_mutex 잠금안에서 일어남.
이렇게 하지 않으면, pop_head()의 호출은 get_tail() 호출과 head_mutex의 잠금사이에서 stuck 될 수 있음

쓰레드들이 try_pop()을 호출할때 바로 잠궈야 하기때문에,
다음 방법으로 initial 쓰레드를 보호해야한다.

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

**broken scenario**

```cpp
    std::unique_ptr<node> pop_head() {
        node* const old_tail = get_tail();
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == old_tail) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }
```

get_tail() 호출이 잠금 밖에 있다.
head와 tail은 initial 쓰레드가 head_mutex에 잠금을 요청할때 변경된다.
반환될 head 노드는 큐에서 사라진다.
 => head.get() != old_tail일 경우 (큐에 더미노드를 제외한 노드가 있음)

**여기서 문제점은?**

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

## broken scenario

```cpp
    std::unique_ptr<node> pop_head() {
        node* const old_tail = get_tail();
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == old_tail) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }
```

get_tail()을 호출하고 잠금이 시작되기 전에 아이템이 추가되면 문제 발생

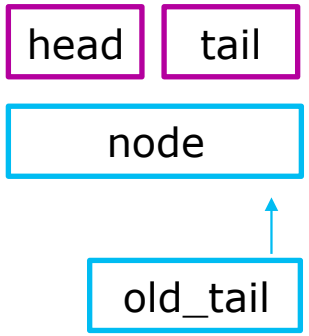
```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

## broken scenario

```cpp
    std::unique_ptr<node> pop_head() {
        node* const old_tail = get_tail();
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == old_tail) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }
```

get_tail()을 잠금 밖에서 호출하고 있다.
첫 쓰레드가 head_mutex에 잠금을 요청할때, 다른곳에서 head와 tail을 수정해버리면, old_tail 노드가 더이상 tail 노드가 아니고, 리스트의 노드도 아니게됨



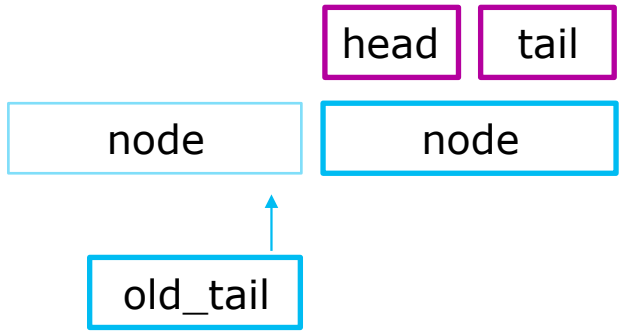
```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

## broken scenario

```cpp
    std::unique_ptr<node> pop_head() {
        node* const old_tail = get_tail();
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == old_tail) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }
```

get_tail()을 잠금 밖에서 호출하고 있다.
첫 쓰레드가 head_mutex에 잠금을 요청할때, 다른곳에서 head와 tail을 수정해버리면, old_tail 노드가 더이상 tail 노드가 아니고, 리스트의 노드도 아니게됨




gg

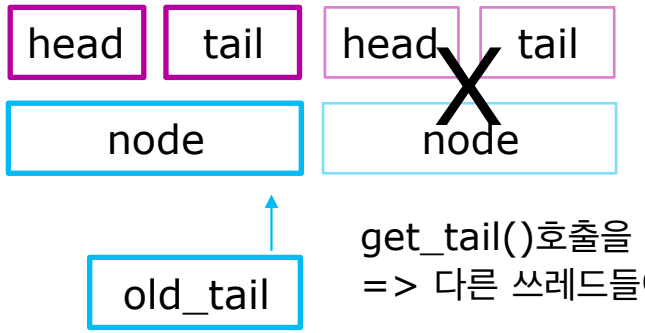
```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

## broken scenario

```cpp
    std::unique_ptr<node> pop_head() {
        node* const old_tail = get_tail();
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == old_tail) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }
```

get_tail()을 잠금 밖에서 호출하고 있다.
첫 쓰레드가 head_mutex에 잠금을 요청할때, 다른곳에서 head와 tail을 수정해버리면,
old_tail 노드가 더이상 tail 노드가 아니고, 리스트의 노드도 아니게됨




get_tail()호출을 잠금 안으로 옮겨야 한다.
=> 다른 쓰레드들이 head와 tail을 수정하지 못하고 invariant 상태가 유지

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

## broken scenario

```
std::unique_ptr<node> pop_head() {
    node* const old_tail = get_tail();
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if (head.get() == old_tail) {
        return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
}
```

```
std::unique_ptr<node> pop_head() {
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if (head.get() == get_tail()) {
        return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
}
```

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
std::unique_ptr<node> pop_head() {
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if (head.get() == get_tail()) {
        return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
}
```

```cpp
std::shared_ptr<T> try_pop() {
    std::unique_ptr<node> old_head = pop_head();
    return old_head ? old_head : std::shared_ptr<T>();
}
```

pop_head()가 head를 업데이트하면서 노드를 삭제하고 나면, mutex의 잠금이 풀림
 => try_pop()은 데이터를 꺼내고, 삭제해 버린다.

안전함 문제될거 없음

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
std::unique_ptr<node> pop_head() {
    std::lock_guard<std::mutex> head_lock(head_mutex);

    if (head.get() == get_tail()) {
        return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
}
```

```cpp
std::shared_ptr<T> try_pop() {
    std::unique_ptr<node> old_head = pop_head();
    return old_head ? old_head : std::shared_ptr<T>();
}
```

try_pop()에서 뮤텍스를 잠그는 부분에서 예외가 발생할 수 있다.
잠금을 걸기전에는 데이터를 수정하지 않는다
 => try_pop()은 exception-safe 하다.

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

# Listing6.6 세밀한 잠금이 적용된 쓰레드세이프 큐

```cpp
void push(T new_value) {
    std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
    std::unique_ptr<node> p(new node);
    node* const new_tail = p.get();
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    tail->data = new_data;
    tail->next = std::move(p);
    tail = new_value;
}
```

push()는 잠그기 전에 하는 연산이 있다.
 => 위에서 예외가 발생하면?

스마트 포인터를 사용하기때문에 예외가 발생하면 알아서 해제된다.

예외가 발생했을때 잠금이 걸려도, 잠금 후에 할 작업이 없음
 => push()또한 exception-safe하다.

## Listing6.6

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

## Listing6.2

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

자료구조 설계자로서 스스로에게 물어봐야할 질문들

- 잠금의 범위를 연산의 일부만 잠그도록 제한할 수 있는가?
- 자료 구조의 각 부분마다 다른 뮤텍스로 보호받을 수 있게 할 수 있는가?
- 모든 연산들이 같은 수준의 보호를 받을 수 있는가?
- 동시성을 위한 기회를 최대화할 수 있는가?

## Listing6.6

```cpp
#include <memory>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };

    std::mutex head_mutex;
    std::unique_ptr<node> head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail() {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    std::unique_ptr<node> pop_head() {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if (head.get() == get_tail()) {
            return nullptr;
        }
        std::unique_ptr<node> old_head = std::move(head);
        head = std::move(old_head->next);
        return old_head;
    }

public:
    threadsafe_queue() : head(new node), tail(head.get()) { }

    threadsafe_queue(const threadsafe_queue& other) = delete;

    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_head = pop_head();
        return old_head ? old_head : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_tail = p.get();
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data = new_data;
        tail->next = std::move(p);
        tail = new_value;
    }
};
```

## Listing6.2

```cpp
#include <queue>
#include <__mutex_base>

template<typename T>
class threadsafe_queue {
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue() { }

    void push(T new_value) {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
/* 1 */ data_cond.notify_one();
    }

/* 2 */
    void wait_and_pop(T& value) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
    }

/* 3 */
    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
/* 4 */ data_cond.wait(lk, [this] { return !data_queue.empty(); });
        std::shared_ptr<T> res(std::make_shared(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value) {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) return false;
        value = std::move(data_queue.front());
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty())
/* 5 */     return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```